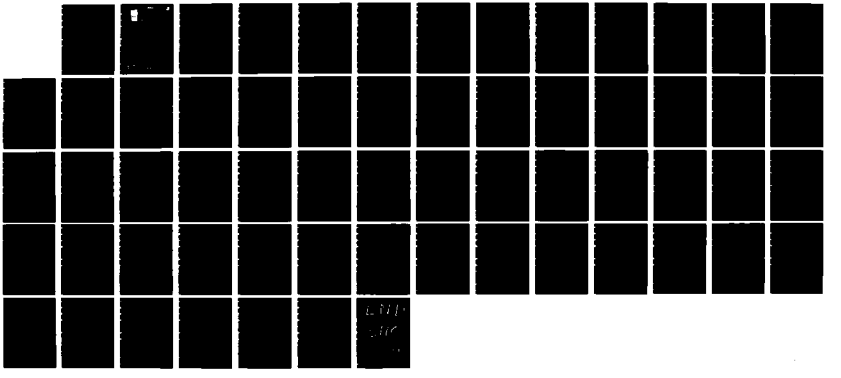
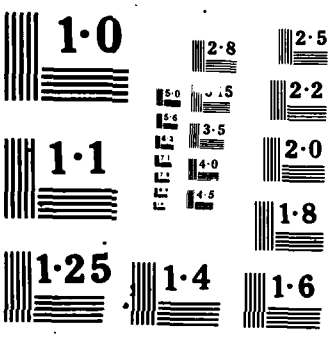
AD-A168 086 REUSABLE SOFTWARE(U) ROYAL SIGNALS AND RADAR 1/1
ESTABLISHMENT MALVERN (ENGLAND) L N SIMCOX ET AL. 1985
RSRE-MEMO-3884 DRIC-BR-99098

UNCLASSIFIED F/G 9/2 NL





RSRE MEMORANDUM No. 3884

ROYAL SIGNALS & RADAR ESTABLISHMENT

REUSABLE SOFTWARE

Authors: L N Simcox, R G Ball

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3884

REUSEABLE SOFTWARE

Authors: L N Simcox and R G Ball

SUMMARY

The issues involved in reusing software are examined both in a general context and in the context of a research environment. In the general context, the characteristics of reusable software are discussed and the views of both the potential developer and those seeking to reuse existing software are considered. An assessment of reusable software is carried out using the concept of the life cycle model, the economic aspects being based on the COCOMO parametric software cost model. General recommendations for improving the portability situation are made for those involved in procuring, developing, and researching software.

CARDPB Ref 2.4.1.1
RSRE Task 424
CAA Sponsor: Director of Data Processing (ATS)
CAA Liasion: Dr J H Crowther RD1
Mr J Scott DP1a

Work done under contract to the UK Civil Aviation Authority

REUSEABLE SOFTWARE

CONTENTS

1. Introduction.

2. Life cycle model.

3. Features of reuseable software.

4. Economics.

5. Implications in a research environment.

6. Summary and conclusions.

7. References.

8. Glossary.

AUTHORS NOTE: This memorandum is a slightly modified version of a report written while investigating how software produced during research might be progressed, where appropriate, through evaluation and further development into an operational system. The more parochial sections have been retained, and a glossary of less well known terms has been added for the convenience of the wider audience.

CHAPTER 1

INTRODUCTION

Much has been written about the serious risks involved in procuring large computer based systems. These risks, which were recognised over a decade ago and are still relevant today, include large time and cost overruns, failure to meet the user needs, shortfalls in reliability and availability, and excessive maintenance costs. The complexity of computer based systems has generally increased over the years with most of the increase coming in the software area, and with hardware costs falling dramatically, this has resulted in the software/hardware cost ratio roughly inverting itself over the last ten years such that now software life cycle costs may well be over four times that of hardware (ref 1).

Software engineering techniques such as high level programming and structured methods have helped improve programmer productivity but not enough to keep up with the estimated demand: Boehm suggests around 5% per annum increase in programmer productivity whereas the US space programme, for example, needs about 20% (ref 4). The Alvey directorate estimate the shortage of skilled staff in UK information technology for 1988 to be around 5000, while the US National Science Foundation estimate a shortage of some 140,000 system analysts and programmers in a similar timescale. Coupled with the procurement risks, this increasing demand for software has lead to the term 'software crisis' and has resulted in the initiation of several government and industry sponsored software engineering related programmes over the past few years. Of particular note are the Alvey and STARTS activities in the UK, and the Dept. of Defense Ada programme in the US (ref 3, which also summarises other major programmes, and ref 5). The general theme of these activities within the software engineering field is the improvement in productivity and maintenance of reliable software through automation and standardisation. Standardisation means cheaper software by such means as reduced training times,

fewer human and technical communication problems, and easier reuse of both application and tool software. Indeed in the rather general sense of the software context here reuseability can almost be thought of as a synonym for standardisation.

Reuse of software is potentially the one way of achieving great economies. Consider the numerous times certain pieces of software, for example text editors and screen text handlers, must have been and still are being rewritten and the attendant waste of effort. In fact, a recent study (ref 22) has suggested that this is true of up to 85% of 'new' software. Therefore, implementors of large systems will look for ways of reusing the software they have already to enable them to offer the lowest bid in competitive tenders, although there is still a tendency not to put in enough effort initially to produce highly portable software. In more quantitative terms, the impact on project cost and time scale can be gleaned from software cost models (see chapter 4) which use the lines 'L' of delivered (new) source instructions as the primary cost driving parameter. For instance from a simple cost model where the manpower is proportional to 'L', it follows that is extremely desirable to employ as much portable software as possible since this would result in zero or at least a small fixed cost.

As well as looking where and how existing software can be reused, it is equally, if not more, important to ensure that, where appropriate, software being developed has those characteristics that will ease future reuse. While manufacturers/vendors apparently do this with items like operating systems and certain application packages they naturally only do so to the extent that it is to their benefit. Even when commercial pressures do not apply it is not always sensible or practical to apply measures to ensure maximum portability. For example in prototyping for feasibity investigations, frequently it will not be cost effective to apply the full rigour of the development process that would normally be applied in a production system since the prototyping may be being used in a rapid turnaround exploratory mode. In such a case the only reuseable element might be just the concepts involved. On the other hand a prototype might be used for developing an algorithm which could lead a reusable design specification and even a piece of reusable code if the object machine environment could be matched.

Furthermore, experience has shown that frequently there are difficulties in persuading system contractors to accept software originating outside their own organisations. This is because of apprehension regarding the effects of trying

to integrate 'foreign' material into their own production lines. Perhaps there is some analogy here with the field of mechanical production engineering. In this there is a long history of experience in the use of 'simple' machine tools such as lathes, millers, planers etc. It is, therefore, possible to introduce an improved such simple tool into the production line and be reasonably confident as to its (beneficial) effects on output. But when it comes to introducing, say, advanced numerically controlled tools, or 'intelligent' robots, there is much less experience to draw on. Thus, it is much more difficult to assess the impact on the output of both the particular part of the line concerned and also on OTHER parts of the production process. Invariably, there seem to be teething troubles which can produce an hiccough in production - though this may not, necessarily, be obvious to customers. At the risk of forcing the analogy a little, compilers, translators and editors can be considered as being roughly equivalent (in EXPERIENTIAL status) to lathes, drills etc; specification languages and verification techniques are on a par with the latest in intelligent robots; debuggers and configuration control methods are roughly in the middle area occupied by NC machines. Also, just as introducing new methods into mechanical engineering may call for a new breed of engineers to operate and maintain them, so, with software engineering, re-education is necessary if new techniques are to be effectively employed. The upshot of this is that contractors are generally only willing to accept EITHER a Specification, from which they can work in their own way, OR a FULLY ENGINEERED, finished module for inclusion in the final product without modification. Any, partially finished, intermediate product they tend to view with mistrust - possibly with good cause!. Those whose business it is to produce software are no exception to this rule - they will have their own methods and (possibly) toolsets and do not wish to get involved in the problems of marrying-in someone else's misfits.

From the above discussion it should be clear that reusing software is a much more complex issue than just 'lifting' a piece of code. First it has to be decided if the issue is the development of software for likely reuse or the reuse of existing software. In the latter case it is necessary to identify which software functions are candidates for reuse and for each selected function which of the life cycle products (specifications, design, code, etc., see chapter 2) can and will actually be reused. Which functions are selected will require analysis of the user requirements and the proposed system design in relation to the candidate software for reuse. An investigation of the system where the candidate software resides will often be

needed. A further analysis will determine cost/time savings which when taken together with other criteria discussed in chapter 3 will lead to the decisions on the extent to which software can be reused.

In the former case, where software is being developed with reusability in mind, the functionality will have been determined either by the actual requirement for the software, or by the potential reuse through analyses of perceived user requirement and/or system design, or by an analysis of the market place for general purpose packages and utilities. Which parts of the life cycle products should be developed specifically with reuseability in mind will require further consideration of the features described in chapter 3.

It is important in making the decisions pertaining to the reusing software that management is able to obtain estimates of the costs involved. Accurate estimation of software cost is notoriously difficult and in this respect software cost models provide the most satifactory general purpose tool for making such estimates. Chapter 4 describes a specific model and how it can be manipulated to help assess the economics of reuse.

Software cost estimation techniques have in the main been developed for use after the system's requirement specifications are known, at least informally. Applying the techniques before this time needs some care, but it is possible to gain insight into the costs and consquences of trying to invoke software 'commercial' development disciplines in a research environment. Because the objectives of research are normally significantly different from those involved in implementing an operational system, the reuse of software developed as part of a research programmme is dealt with separately in chapter 5.

The main concepts discussed and developed in the paper are summarised in chapter 6, and some general guidelines are put forward for the developers, procurers, and researchers to improve the reusability characteristics of the software they produce and adopt.

To set the background to the discussions, a review of the software development process in terms of the concepts and terminologies of life cycle models is presented in the following chapter. As reinforced in that chapter, the term software includes all the documentation and information - such as specifications, design and acceptance tests - relating to the computer program and not just the progam code.

CHAPTER 2

LIFE CYCLE MODEL

Control of a large system requires project management to have a conceptual model of what is being controlled. Software engineering uses the term 'life cycle model' for such a model. There are numerous models in use, the simplest of which defines four sequential phases - requirement, design, production and maintenance (ref 18). The phases and their names vary from one model to another, and even the same name in different models may not have precisely the same meaning.This situation is summarised in table 2.1 which compares the phases and their definitions for several life-cycle models over a period of time corresponding from approximately the end of the main research phase to handover to the customer. The last column in this table is the model more fully elaborated below.

The representative model used here follows closely that described in the STARTS guide (ref 5). The main differences are in the addition of concept, feasibility and project definition phases before actual project authorisation. This has been done in order to address the R&D effort and the work done on the user requirements before the implementation proper begins: The STARTS guide model is very much orientated towards software development after the request for bid and contract let stage, and neglects the earlier phases.

Each phase is defined by its output rather than its work activities and for the representative model the phases are described below :-

1 Concept. Project concepts are explored often in association with research and development. Typically the successful output would a Staff Target in military terminology or a request for an evaluation in the NATS case.

2 Feasibility. This phase is likely to involve feasibility studies, further R&D, and evaluation trials. The aim is to refine the project concept. First reliable estimates of software size and cost are available. The high level User Requirement is produced.

3 Project definition. Further studies and experiments may take place. Outline project plans are produced. User Requirements are developed into a set of Procurement Specifications. These together with other contractual documents form the basis of the request for proposal (RFP) package.

4 Initiation. A bid proposal is made which includes a top level project plan with milestones, resources, schedules, etc.

5 Requirement specification. The procurement specifications are worked on to produce a complete ,validated specification of required functions, interfaces and performance for the software product.

6 Structural software design. A verified design of the overall software is made.

7 Detailed software design. The detailed software components and structures are specified and verified.

8 Code and test. Software components are programmed and tested.

9 Software integration and test. Software components are integrated, tested.

10 System integration and test. Software and hardware is brought together to produce a fully operational system. This includes customer acceptance testing, training and documentation.

11 Maintenance (Evolution). This phase includes corrective, restructuring and enhancement activities.

12 Phaseout. Transfer of functions to a replacement product.

The model is idealised, thus in real software systems for example,

- not all phases may be present,
- the phases are not strictly sequential since

exploratory work on a subsequent phase will normally be carried out before the given phase is complete,
- different parts of the product will be in different phases at any given time,
- the users perceived requirements tend to change,
- prototypes may be employed in more than one phase, and some models treat protyping as a separate phase (ref 5),
- reworking of earlier phases may be necessary as a result of problems encountered in later phases: iteration is common in real system development,
- the definition of the actual project phases may differ markedly from the model.

It should also be born in mind that a project's work breakdown will group work tasks into specific activities such as project management or quality assurance; confusingly the name given to an activity is often the same as that given to a phase. A good example of this confusion is seen in the activities defined in the STARTS guide (ref 2) :- Project management, Requirement specification, Structural Design, Code and Unit Test, Test Planning, Verification and Validation and Testing, Quality Assurance, Configuration Management, Manuals. In general each of the activities will contribute effort towards the completed products of a given phase and it is common to show this by means of a matrix of activities against phases (refs 2 and 5).

Table 2.1

Comparison of life-cycle models.

DoD-USAF ref 14	BOEHM ref 13	BOEHM ref 1&2	METZGER ref 15	FREEMAN ref 16	STARTS ref 5	Defined in This chapter
SYSTEM REQU'TS	//////////	//////////	DEFINITION	NEEDS ANALYSIS	//////////	PROJECT DEFINIT'N
	SYSTEM REQU'TS	PLANS & REQU'TS			INITIAT'N	INITIAT'N
	SOFTWARE REQU'TS			SPECIFI-CATION	REQU'TS SPEC.	REQU'TS SPEC.
	PRELIM-INARY DESIGN	PRODUCT DESIGN	//////////	//////////		
SYSTEM DESIGN			//////////	//////////	STRUCT-URAL DESIGN	STRUCT-URAL DESIGN
PRELIM-INARY DESIGN			DESIGN	ARCHITE-CTURAL DESIGN		
DETAILED DESIGN	DETAILED DESIGN	DETAILED DESIGN		DETAILED DESIGN	DETAILED DESIGN	DETAILED DESIGN
CODE PRODUC'N	CODE & DEBUG	CODE & UNIT TEST	PROGRAM-MING	IMPLE-MENTA-TION	CODE & UNIT TEST	CODE & UNIT TEST
INTEGR-ATION & VALIDA-TION	TEST & PREOP-ERATIONS	INTEGR-ATION & TESTING	SYSTEM TEST		INTEGR-ATION & TESTING	SOFTWARE INT'N & TESTING
			ACCEPTANCE			SYSTEM INT'N & TESTING

CHAPTER 3

FEATURES OF REUSEABLE SOFTWARE

3.1 GENERAL CONCEPTS

One's view of reusable software is very much coloured by whether one is trying to produce software for subsequent use or whether one is looking for existing software for reuse. In both instances, one is trying to treat a software product output from a life cycle phase in one system as the an input to a phase in another system. The diagram in figure 2.1 shows this concept in an idealised manner. The software in each is represented by a set of life cycle products grouped by phase. A software product P in system 1 is identified for reuse in system 2 and is suitably modified to become product P'. Remember that the term software encompasses all the life cycle products, hence P is not necessarily program code.

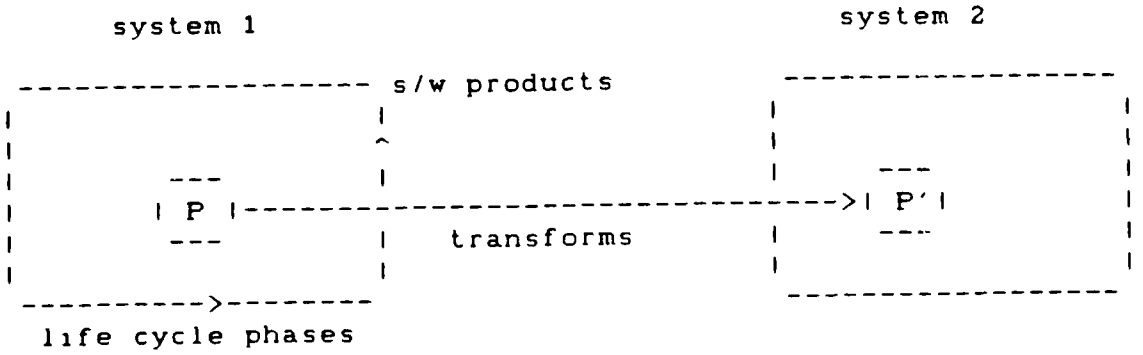


Figure 3.1

Reuse of s/w product P

FEATURES OF REUSEABLE SOFTWARE

If a software product is reusable without modification it is said to be portable. In practice this strict definition is relaxed to allow trivial and sometimes minor modificatons.

The work activities mentioned in the previous chapter can be incorporated into the simple software model used in figure 2 by drawing a third axis perpendicular to the plane of the rectangle. The software P will then have 'dimensions' in this third axis so that moving a software product from system 1 to system 2 will in general involve all the work activities to some lesser or greater degree. Thus even portability does not come free.

A further complication arises because the outputs of each phase are dependent on the inputs of the previous phases, and on the iteration processes due to subsequent phases. This comes as no surprise since, for example, reuse of a piece of code will normally involve reuse of some specification, design and test products.

A number of key questions arise in trying to apply this abstract model of software:

1. How are the products P and P' identified?

2. How is P transformed into P'?

3. What is the decision process in authorising the reuse?

When we discuss these questions in the context of specific development scenarios we will find that several other important issues are raised including that of hardware dependency.

3.2 REUSE SCENARIOS.

The scenarios we are interested in fall naturally into two categories:

1. A project is seeking to reuse or adapt software from another system or systems.

2. A project is developing software, with or without serious consideration of reuse.

In practice most software development exhibits the characteristics of both categories to some extent, although it is convenient to treat the categories as separate aspects of the development process for analysis purposes. A more detailed examination of the categories follows.

3.2.1 Category 1.

All systems fall into this category in the special sense that they import some of the knowledge built up from three decades of computer technology, and in the sense that porting of system software (as opposed to application software) in the form of operating systems, compilers and similar items will nearly always be involved. In the software application area, adoption of published algorithms is a good example of reuse, which often require a translation from say Fortran to Pascal. Portability in the project definition phase often occurs when a procurement agency reuses part of one Bid Request package, such as its requirements for software standards, in the Bid Request documents of another procurement. Many of the problems likely to be encountered in trying to reuse existing software are seen in the following case study.

3.2.1.1 Case Study - A large on-line information management system is to be procured. The system must receive and transmit both data and command messages to local and distant terminals. High integrity, security, and availability are required. User terminal facility will include message handling, database monitoring using graphics and a query language, and applications involving non-trivial calculations on database items.

The project is nearing the end of the Project Definition phase and the required user's functions and system performance has been determined. Estimates of the software, database, and computer sizes have also been made with the aid of a high level system design. The procurement agency is aware of a system which already exists that performs user tasks similar to those required by its system, and is eager to reuse any software products. Although access to the life cycle product of this similar system is possible, unfortunately the procurement policy demands that an open tender situation shall exist and that the procurement specification documents shall not specify hardware or software that would show preference for any particular tenderer. Thus at this stage portability of code had to be ruled out since common programmming langauges and run time environments could not be

assured. Thus the most that could be expected was reuse of some requirement and design specifications, possibly with some reuseable code that would need translating to another language.

The method adopted to assess which products should be reused consisted of:

1. A comparison of the two systems to identify those user functions that were sufficiently similar to one another to justify further analysis at the design level.

2. The software that performed those functions identified in the existing system was examined and the technical feasibility of reuse was assessed.

3. Where reuse was deemed technically feasible the cost of converting the software into a useable product was compared with that of developing the same product from 'scratch'. Only where a significant economic advantage was clear was reuse authorised.

The features of technical feasibility addressed included:

System dependence - To what extent is the software dependent on any hardware or software features of the machine on which the system is currently implemented?

Data dependence - To what extent is the software dependent on specific representations of data and how difficult is it to modify the data representation so that the software can be reused?

Modularity - How easy is it to isolate modules/sub modules and to remove them? How easy is it to insert new modules/sub modules? Does the software have clear and well defined interfaces?

Integrity/security - Does the software provide the required levels of of integrity and security? How difficult is it to add or amend the integrity and security controls? What level of verification and validation was applied and is it sufficient for the new application?

Performance - Does the software and hardware provide the required response time, data base capacity, expansiblity, reliability etc?

Design - If it is not feasible to reuse the code what parts of the software design can be captured and will the design tools be available in the new system?

Documentation - Is the documentation sufficient and accurate enough to permit extraction of software and subsequent amendment? Can the software be maintained by other than the original development system?

The main driving factor throughout the study was that of cost and if, at any stage during the analysis, it became clear that reuse would not be cost-effective the appropriate piece of software was removed from the short list. The cost of scratch software was based on a six phase life cycle with costs apportioned to each phase according to the table 3.1 (apparently taken from a Boeing cost model, cf page 518 of ref 2). For comparison purposes, the table also gives the approximate correspondence of the phases to those defined for the representative life cycle model in chapter 2. Of the several items found that were technically reuseable only a few could be adapted economically. Even then it was recognised that it would not be possible to save anywhere near the development costs indicated in table 3.1. Firstly, because only products from the requirements and design phases were being considered for reuse, ie the maximum saving for the requirements and design phases from the table is 30% and, secondly, the advice from software developers was that some 65% of the effort in these phases would still be needed before coding could begin. Thus cost of the software authorised for reuse would be around 90% (70+30*0.65) of the originally estimated value, a saving of only 10%.

Table 3.1

Costs of software by phase

representative phase no. (see chapter 2)	phase name	% cost
5	req' def.	5
6,7	design spec.	25
8	code	10
8	code & test	25
9	s/w int. & test	25
10	system integration	10

The lessons here are clear. Reusability may be not as easy or economic as one may think, and considerable effort may be needed to identify reusable components and convert them into the required form. In this particular exercise two of the major features, common hardware and common software development environment, contributing towards portability were missing. Had the procurement agency been able to specify these features it would then have been found that reducing the costs of the current procurement through a preferential specification would result (in this instance) in the user necessarily having different hardware and software from his (the user's) existing systems; thus possibly leaving the user with no overall economic gain. A further problem with directed procurement is that the user's requirement may have to be 'bent' so much in trying to make it compatible with the existing product that any cost saving he makes will be wiped out enhancing and modifying the system to work in his environment: he may also have saddled himself with an outdated and outmoded system.

The techniques of emulation and preprocessing can be very worthwhile in achieving reuse in some instances. For example, the machine instructions of one computer may be emulated by the microcode of another, by its software or by a combination of both. This provides a simulation of the processor, but differences in features such as speed, storage capacity and peripherals are likely to prevent pure portability of software. However, a full emulation can be a convenient route for updating to more modern equipment without changing any software. Preprocessors enable reuse by translating from, say one dialect of a high level language to another. Here again, the differences in hardware and underlying software (eg. operating system) mean that manual amendments are needed before a program can run correctly.

3.2.2 Category 2.

It is convenient to grade developmemt of software with reuse in mind into different levels according to the degree of portability being built into the product. Three levels are considered sufficient here:

1. The aim is to achieve portability.

2. There is a high motivation for producing software that is easy and simple to reuse.

3. There is little or no motivation directed towards

reuse.

In the first level one tends to find the commercial software products such as operating systems and compilers, and application packages such as mathematical equation solvers. It is interesting to note that portability is achieved through a layered approach with standardised interfaces - operating systems rely on standard hardware, compilers on operating systems, application packages on compilers and operating systems. Other software widely reused includes various international, national and de-facto standards such as high level programming language like Fortran, and software development methods and tools such as the MASCOT method.

Most new software specially developed for an operational system tends to fall into the second level. However because of the pressures of cost and timescales there is usually no specific effort directed towards reusable products.

In the third level are those experimental systems and rapid prototypes which are concerned with the testing and broadening of ideas and concepts.

3.3 CHARACTERISTICS OF REUSABLE SOFTWARE.

It is appropriate to concentrate on the characteristics of portable software since some or all of those characteristics will be needed in achieving a given level of reusability. It is perhaps easiest to identify these characteristics by considering a computer program in the abstract as shown in figure 3.2. References 23 and 24 review other, more formalised software abstraction constructs (including flow control abstraction of structured programming and the abstraction concepts of Ada) and their relevance to reducing complexity and enhancing understanding. A program will carry out some task to meet a functional specification. As a direct consequence of the functionality requirements a program will need to have access (read and write) to data and must correctly interface with other programs (calling and being called). It will also lie in a hardware/software environment which can affect the desired behaviour of the program, particulary in regard to speed of operation.

In essence the diagram represents the template of an abstract program specification. The specifications will tend to be informal and fuzzy during the earlier life cycle phases and become more formal and mathematically precise as

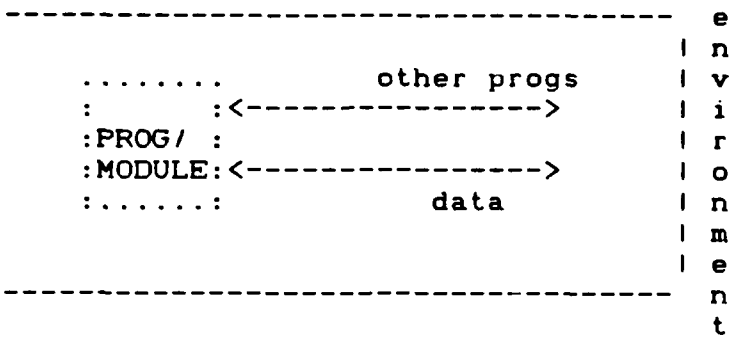


Figure 3.2

Abstract program specification template

coding is approached. There will also be functional and design decompositions as the software development progresses within and through the life cycle phases, resulting in a number of modules each of which have the basic structure depicted in the figure.

A guarantee of software portability from one system to another clearly imposes some very severe constraints on the software and/or on the systems involved. Firstly, the interfaces with data, other software and the environment will need to be identical in the two systems. It is extemely difficult to specify completely and precisely these interfaces and unless identical systems/subsystems are being produced it may be impossible to achieve pure portability. Secondly, the software in the producer system must have been validated and verified, and there must be some easy mechanism to enable the physical transfer of the relevant software to the new system. However, in spite of these difficulties, by adopting appropriate measures the software developer can ensure the potential for economic reuse can be made quite high. Such measures will come under one or more of the following headings:

1. Functional decomposition to produce general purpose subfunctions.

2. Design decomposition to seek modules with simple interfaces and simple interconnectivity. Separation of hardware dependent software.

3. Use of standards, tools, and methods.

4. Mature, proven, and widely used standards.

5. Reliability assured by proper verification and validation.

6. Library of reusable components.

7. Up to date documentation.

8. Correct, complete and unambiguous documentation.

9. Management commitment.

The way that software is broken into subfunctions and modules strongly affects the potential reuse. At the functional level, a program that performs general related tasks is more likely to find a customer than one which performs rather specific unrelated tasks. For example a program that outputs data to a terminal in only one format is less attractive than one which permits user definable formats. Modular design also strongly influences the quality of software and, properly carried out, should reduce complexity and improve reliability and maintainability. A number of methods and concepts have been developed to assist in the modularisation process: there is some current research work aimed at unifying the leading methods (ref 9).

Adoption and adherence to standards is the backbone of portability. Standards are relevent to many aspects of software development, including development methodologies, tools and methods, and documentation. A methodology is a systematic set of procedures which define the managerial and technical methods to be used in such areas as documentation, analysis, design, coding, testing, maintenance, and configuration management. It is acknowledged that the methodology is pre-eminent over the tools used and must consider the overall system including hardware and environmental factors. Thus a methodology should include those guidelines to be followed in regard to reusability aims.

While tools and methods are available to provide assistance in all the life cycle phases, they are most mature and abundant in the detailed design and coding phases. Tools in the typical software development environment will include operating system, compilers, editors and filing systems. A survey of the less commonly used tools such as specification languages, cost models, and code analysers has been carried out under the auspices of the DTI and is published in the STARTS guide (ref 5). In recognition of the need to integrate the important tools, bodies such as Alvey and the US DoD have set up programmes to produce Integrated Program Support Environments (IPSE or

PSE). The standards which will be imposed when using these PSEs will ease reuseability problems, particularly since one aim is to increase target-computer independence.

The main language in both the UK and US PSEs referred to is Ada (ref 7). The adoption of Ada, with its strictness of definition, its prohibition of subsetting and supersetting, and the formal requirement for validation of Ada compilers, together with Ada being an international standard and being forced on the US military, should eventually lead to a greatly improved portability/reuse situation: indeed the Ada programme arose as a result of investigations into the current and projected cost of software and the economies of software reuse. The commonly cited technical features of Ada which support reusability are (refs 23 and 7):

1. A rich variety of program units (modules) such as subprograms, packages, and tasks.

2. Systematic separation between visible syntactic interface specifications and bodies. This separates the concerns of interconnection from how the module performs its functionality.

3. Strong typing which imposes checks on module connections and parameters.

4. Generic programming units, i.e. parameterised templates for generating software components.

5. Program libraries.

Even with Ada, complete target machine independence is not always possible, although by adopting some mandatory rules and guidelines, portability or near portability of non time-critical software can be achieved (ref 8).

Software for reuse must be reliable. The degree of reliability that is acceptable will vary according to the system and component involved. A compiler which fails to compile an infrequently used but legally correct construct is more acceptable than one which compiles illegal source code or produces incorrect object code without flagging a fault. The more critical a project is in terms of such factors as safety and security the more confidence the system developer must have in the the software reliability. This confidence can be gained by having access to verification and validation test results and where necessary repeating and extending the tests.

FEATURES OF REUSEABLE SOFTWARE

In order to provide potential users with an awareness of the availability of reuseable software components software libraries are needed, and where reuse is seen as a major cost saving possibility, large and complex software libraries are being advocated (ref 12). These libraries would have to contain tools to assist in integrating components together, testers, debuggers, and analysers. A library would also contain an element called the 'property data base' which would include information such as the 'knowledge' of the development and maintenance processes of a component.

There are some difficult problems to be solved in producing a software component library which is efficient, reliable and convenient for a reasonably wide range of applications:-

How can one determine which components are generally useful and capable of being specified?

How does one specify the components so that they be can understood and used by others?

How are the components to be catalogued?

Note that the work required to build and maintain such libraries might mitigate against the potential advantages of reuse.

In the literature software libraries are discussed usually in terms of the context of code components, however more time is spent on the requirements, design, and maintenance phases than on coding so research should also address these areas. Thus much of the documentation related to portable software could be kept in the software library database. By ensuring the consistency and timeliness of updates to the library the needs for accurate and up to date documentation could also be met. The functionality, interfaces and test history of a software component could also be provided as accessible items which could add to the understanding and confidence needed by potential users.

Adoption of measures to achieve a given degree of portability will need management commitment to them and to ensuring that the measures are effectively applied during development and maintenance. A strong commitment is essential because considerable extra resources may be required to reach the desired portability requirements. The economics of the situation is discussed further in the following chapter.

CHAPTER 4

ECONOMICS OF REUSE

4.1 ESTIMATING METHODS.

A quantitative assessment of the economic aspects involved is essential to any decision concerning the practicalities of resuseable software products. An example of such an assessment has been given earlier (section 3.2.1) in a case study of a proposed system seeking to use software from an existing and apparently similar system. The assessment was made there by using a life cycle model to identify software products by phase, and to cost these products using historic data that apportioned software project development costs according to phase. Only part of this potential cost saving was possible because of the effort needed to convert the products for the proposed new system. The potential cost saving 'S' in such cases is given by

$$S = \text{Cost of new implementation} - \text{cost of reimplementation} \quad (4.1$$

This difference can be expressed as a percentage of the cost without reuse,

$$\frac{S * 100}{\text{Cost of new implementation}} \quad (4.2$$

In fractional form, this expression ranges from 0 to 1 and is known as the degree of portability: the assumption being that it is cheaper to reuse/adapt existing software than start again - an assumption not necessarily true.

The cost of reimplementation can be reduced by building in portabililty features discussed in the previous chapter. This will in general increase the initial cost of implementation by a amount C, where

C = cost of implementation with portability features - cost without.

This extra cost can be justified by amortizing it over several reimplementaions where the same Organisation is involved, or by spreading cost over the whole life-cycle, or by profits to be made by licensing /selling the product in the market place. In the latter instance the effective cost of reuse to the purchaser will be roughly the cost of the licence fee (provided it meets the requirements) plus the costs of integrating it with the other software.

There are various ways of classifying cost estimating techniques (refs 11 &17) of which the main ones are considered to be:

(i) Cost models: these provide various formulae which produce cost estimates as a function of one or more parameters.

(i) Bottom up: the software is split up into individual components, each costed separately by some responsible person. This method needs some design effort and may overlook system wide costs.

(ii) Analogy: this method involves making comparison with completed projects.

(v) Expert judgment : this involves consulting one or more experts and is prone to bias.

No one method is consistently better than the others in all circumstances and table 4.1 taken from refs 2 and 11 indicates why this is so in terms of some of their strengths and weaknesses.

The combination of a cost model and a bottom up approach would seem to offer the basis of a satifactory method. Since both techniques need experience and expertise to be effective, such a combination would in effect involve elements of all four techniques listed. Which ever cost model is adopted, there wil be a sequence of actions to follow which would be similar to those steps identified by Stanley (ref 17):

1. Estimate software size.

2. Convert estimate to labour and money.

3. Adjust estimates for unusual characteristics.

4. Divide estimates into the the different project

Table 4.1

PROS and CONS of COST ESTIMATION METHODS

Method	Strengths	Weaknesses
Cost model	objective repeatable sensitivity analysis calibrated	inputs often subjective unusual characteristics not considered but calibration data may be out of date
Bottom up	detailed stable fosters individual commitment	may overlook system wide costs requires more effort
Analogy	based on experience	past experience may not be representative
Expert judgment	copes with unusual characteristics	subject to bias *no better than experts*

phases.

5. Estimate non technical labour and computer time costs.

6. Sum the costs.

Not all steps may occur and steps may be performed in a different order or in parallel. For example a bottom up approach may involve producing estimates of software components by phase, hence step 4 would aggregate the separate estimates in parallel with estimating the overall software size in step 1.

While it is not possible to compare or even review the different cost models, many can be summarised by two equations which relate the amount of effort 'E' needed to develop the number of source code line 'L', and the time 'T' in which to do it. The equations are

$$E = a(L)^{b} \quad \text{and} \quad T = c(E)^{d} \qquad (4.3$$

where a,b,c, and d are parameters derived from historic data bases and characteristics relating to the software to be developed, the precise derivation depending on the particular model. The units normaly adopted are the ones also adopted here, namely man months, thousands of lines, and months for E,L, and T respectively. For typical values of a,b,c,d equal to 3.0, 1.10, 2.5, 0.35, and software of 6000 lines of code, equation 4.3 gives

$$3(6)^{1.1} = 21.5$$

man months of development effort, which would take

$$2.5(21.5)^{0.35} = 7.3$$

months to complete, with an average staffing of 3.

The published values of the parameters, a to d respectively, range from 1.0 to 5.0, 0.90 to 1.50, 2.5 to 4.6, 0.25 to 0.36 depending on the particular model, the parts of the life cycle phases and work activities covered and to what depth, the exact definition of the quantities E,L,T, and the historic data base used to calibrate the parameters.

To be of use in the present context we need a model that is sensitive to those features that enhance portability and reuse as discussed in the previous chapter. The model must also be capable of estimating the costs of the the separate life cycle products since it is not always just the code that is required for reuse. The most widely used models in the USA appear to be the RCA PRICE model (refs 17 & 18) and the Putman SLIM model (ref 18) - there is little evidence of any widespread use of cost models in the UK. Of the two, the PRICE model seems more appropriate since it does require input of the percentage of the design and of the code which will be new. However, being commercial packages, the details of the internal workings of either model are not easy to come by and mainly for this reason we shall concentrate on the COCOMO model of Boehm which is freely available in reference 2.

4.2 THE COCOMO MODEL.

The COCOMO (Constructive Cost Model) is in reality 3 models - Basic, Intermediate and Detailed. The Basic model provides for quick and rough order of magnitude estimates. The Intermediate model includes additional factors to address the over simplification of the Basic model and to take account of project specific properties. The Detailed model extends the Intermediate model to allow the influence of the addtional factors to depend upon the life cycle phase. The Basic and Detailed models are sufficiently simple that they can be described here in enough detail to show their applicability in estimating the costs of software reuse. The important assumptions underlying all three COCOMO models are:-

1. The development period covered by the equations of the type 4.3 is from the begining of the product design phase to the end of the acceptance phase (see table 2.1). Costs of the other phases are estimated separately.

2. Indirect costs such as that of higher management and secretaries are not included.

3. The delivered number of source lines L excludes comments and unmodified software, however procedures to take account of adapting existing software are given.

4. The influence of cost drivers is phase independent for the Basic and Intermediate models.

5. In any one model the parameter set a,b,c,d in equations 4.3 takes on different values according the mode of software development. Three modes are defined:

ORGANIC: small teams develop software in a highly familiar environment, not applicable with L> 50k lines of code. Examples of products are simple inventory controls and familiar compilers.

SEMIDETACHED: intermediate between organic and embedded, not applicable with L> 300k lines of code. A typical product would be a new operating system or simple command and control system.

EMBEDDED: an environment for developing software which is for systems with strongly inter-related hardware, software, regulations and operational procedures. Large operating systems and complex command and control systems are examples.

4.3 BASIC COCOMO

The parameters for the effort and schedule equations for Basic model development modes is given in table 4.2 below.

Table 4.2

PARAMETERS for BASIC COCOMO

MODE	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

The small values of a and b for the organic mode reflect the fact that small teams can develop a small amount of software quickly and efficiently, while the nearly constant values of c and d show that the development time is insensitive to the mode of development. The proportion of effort E per phase is given in table 4.3. Because the model does not include the Plans & Requirements phase in the calculation of the effort E from equation 4.3, the effort in this phase is estimated seperately as percentage increase of E as given in the table.

Boehm gives tables for the proportion of development time per phase, which can be recast into a set of formulae similar to those given in the table for the development effort.

The steps in using the Basic model to estimate the direct labour costs are:-

1. Determine the development mode.

2. Estimate the number of source lines.

3. Calculate the overall development effort E using equations 4.3 and table 4.2.

4. a) Work out the proportion of effort per phase using table 4.3.
   b) To include the effort in the Plans and Requirements add in the percentage according to

Table 4.3

% EFFORT by PHASE

MODE	PHASE	%
Organic	Plans & requirements	add 6%
	Product design	16
	Detailed design	26 - s
	Code & unit test	42 - 2s
	Integration & test	16 + 3s
Semi-detached	Plans & requirements	add 7%
	Product design	17
	Detailed design	27 - s
	Code & unit test	37 - 2s
	Integration & test	19 + 3s
Embedded	Plans & requirements	add 8%
	Product design	18
	Detailed design	28 - s
	Code & unit test	32 - 2s
	Integration & test	22 + 3s

where s = 0,1,2,3, etc are according to the number of lines of code up to 2k,8k,32k,128k, etc.

table 4.3.

5. Convert effort per phase into cost per phase according to current unit manpower cost per phase.

6. Add up costs per phase.

7. Add other overheads not included (see section 4.6).

Other costs, such as taxes, are added in separately and are discussed in section 4.6. For the maintenance phase, which excludes major redesign and redevelopment but includes the corrective, perfective, and adaptive elements, the annual cost is based on the the amount of code that is

expected to be affected. If this amount is '1', then the proportion of delivered code affected is 1/L and annual maintenance effort E, is estimated using the equation

$$E_M = E * (1/L) \qquad (4.4$$

where E_M is the overall development effort.

4.4 THE INTERMEDIATE COCOMO MODEL.

The Intermediate model extends the Basic model to enable factors which have significant effect on cost to be taken into account. Boehm identifies fifteen such factors which he refers to as cost drivers, and these are listed in the left hand column of table 4.5. Associated with each cost driver in the table is a set of ratings, ranked from very low to extra high, which are used to determine a multiplier to be applied to the effort equations. For example, if it is decided that the software development will make a slightly more than normal use of software tools and a high rating for this cost driver is chosen then the appropriate multiplying factor from the table is 0.91, ie there is a reduction of 9% in the estimated effort. More precisely if the multiplying factor for the i-th cost driver is k_i then the effort equation is

$$E = K * a(L)^b, \qquad K = k_1 * k_2 * \ldots * k_{15} \qquad (4.5$$

K is the product of the all the effort multipliers. In using these multipliers Boehm found it necessary to employ slightly different values of the parameter 'a' from those used the Basic COCOMO, and these new values are given in table 4.4 below.

To use the Intermediate model, the same sequence of steps described for using the Basic model can be followed except that at step 3, equation 4.5 is used instead of equation 4.3 in order to take account of cost drivers.

Instead of merely applying the same cost driver multipliers over the whole of the software, different multipliers can be applied to different components. This is useful particularly where the software consist of several large packages of differing complexity being developed by

Table 4.4

PARAMETERS for INTERMEDIATE COCOMO
(values different from Basic model in italics)

MODE	a	b	c	d
Organic	*3.2*	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	*2.8*	1.20	2.5	0.32

more or less independent teams. For this component level the effort equation is

$$E = E_1 + E_2 + \ldots , \qquad E_j = K_j * a(L_j)^b , \qquad (4.6$$

where K_j is the product of the multipliers for the j-th component of size L_j.

The annual maintenance effort in the Intermediate model is

$$E_M = K_M * E * (1/L) \qquad (4.7$$

where, as in Basic model, 1/L is the annual proportion of software undergoing maintenance. E is the nominal effort needed to develop the L source lines of code using equation 4.3 and table 4.4. K_M is the product of the cost driver multipliers for maintenance which, apart from the reliability and modern programming practices multipliers, is obtained from table 4.5. Those for reliability and modern programming methods to be used for maintenance phase are given in table 4.6. The use of different values in the development and maintenance phases reflects the pay-off in maintenance.

Table 4.5

EFFORT MULTIPLIERS for COST DRIVERS

	COST DRIVER	RATING					
		V.Low	Low	Nominal	High	V.High	X.High
1	Software reliability	.75	.88	1.00	1.15	1.40	-
2	Data base size	-	.94	1.00	1.08	1.16	-
3	Product complexity	.70	.85	1.00	1.15	1.30	1.65
4	Execution time constraint	-	-	1.00	1.11	1.30	1.66
5	Main storage constraint	-	-	1.00	1.06	1.21	1.56
6	Virtual machine volatility	-	.87	1.00	1.15	1.30	-
7	Computer turnaround time	-	.87	1.00	1.07	1.15	-
8	Analyst capability	1.46	1.19	1.00	.86	.71	-
9	Applications experience	1.29	1.13	1.00	.91	.82	-
10	Programmer capability	1.42	1.17	1.00	.86	.70	-
11	Virtual machine experience	1.21	1.10	1.00	.90	-	-
12	Programming language experience	1.14	1.07	1.00	.95	-	-
13	Use modern programming practices	1.24	1.10	1.00	.91	.82	-
14	Use of software tools	1.24	1.10	1.00	.91	.83	-
15	Required development schedule	1.23	1.08	1.00	1.04	1.10	-

Table 4.6

MAINTENANCE EFFORT MULTIPLIERS
(where different from development)

COST DRIVER	V.Low	Low	Nominal	High	V.High	X.High
Reliability	1.35	1.15	1.00	0.98	1.10	-
Modern programming practices product size						
2K	1.25	1.12	1.00	0.90	0.81	-
8K	1.30	1.14	1.00	0.88	0.77	-
32K	1.35	1.16	1.00	0.86	0.74	-
128K	1.40	1.18	1.00	0.85	0.72	-
512K	1.45	1.20	1.00	0.84	0.70	-

4.5 ACCURACY

A knowledge of the accuracy of the COCOMO model, or indeed of any other model or method, is essential for a satisafactory assement of the economics of software development and reuse. Boehm states that the Intermediate model development estimates are within 20% of the actuals 68% of the time when used in its domain of calibration: the Detailed model does not do significantly better overall but may be more suitable where cost drivers are very phase dependent. Thus, while the model as it stands could be used as a black box to aid management decisions, this should be done after assurance that the domain of application is similar to that used to calibrate the published version. If need be the parameters can can be recalibrated by the methods described in reference 2 using historic project data pertinent to one's own software development environment.

It should also be remembered that the COCOMO model does not include clerical effort which is stated to run at about 3 to 4% of the COCOMO cost estimates. The main reasons for not including this in the COCOMO model are that such costs are frequently bundled in with the developer's over head rate and that trying to account for phase dependency would overly complicate the model. Other special to project items will be needed to be added in separately - these items will include the purchase, licensing, rental and maintenance of software tools, computer and travelling time, profits, and taxes. The cost estimates from the model are in units of man-months. While this provides a stable measure of currency, convenient when comparing estimates from other models at different times, eventually it will be necessary to convert to prevailing rate for the job. A rough conversion can be made by using a typical 1984 rate of £3333 per man-month (£40,000 per annum) and adding in costs for items mentioned above. On the total less VAT, a contractor will often add a 'profit margin' of around 10%. A more accurate conversion can be obtained by apportioning effort according to rate charged for the type of personnel involved. For example one might charge consultant rates for the plans & requirements and product design phases, analyst rates for the detailed design phase, and programmer rates for coding and integration phases. Typical 1984 software house rates for consultant, analyst, and programmer are £5400, £2700, and £2200 per man-month respectively.

4.6 USING THE COCOMO MODEL IN COSTING SOFTWARE REUSE.

Both the Basic and the Intermediate COCOMO models can be used to estimate the cost of reusing or adapting existing software or to estimate the extra costs needed to enhance portability features of software to be developed. We consider both approaches in turn.

4.6.1 Adapting Existing Software

Taking the adaptation case first, one technique is to reduce the value of L in the estimating equations by some factor to take account of the fact that it will be cheaper to adapt rather than re-develop software. The technique also acknowledges that it can be more costly to adapt! Boehm suggests a number of methods of doing this. In the simplest one, the component of L representing the software being adapted, is multiplied by an 'adaptation adjustment factor', which is in effect the fraction of effort required for the adaptation summed over the design, coding, and integration & test phases - the design phase is the combined product design and detailed design phases. Taking the typical percentage of development effort for these three phases as:

40% : 30% : 30%

the adaptation factor A expressed as a fraction is then given by

$$A = (0.40 * D + 0.30 * C + 0.30 * I)/100$$

where,

D is the percentage of the adapted software design modified to meet the new requirements,

C is the percentage of the adapted software's code modified to meet new requirements,

I is the percentage of effort to integrate and test the adapted software as compared with that of a similar full development. Boehm acknowledges this may exceed 100 in some situations.

If a project consists of several components of size L(i) with different adaptation factors A(i), the effective value of L to use in the effort and schedule equations is taken as

$$L = L_1 * A_1 + L_2 * A_2 + L_3 * A_3 + \ldots \qquad (4.8$$

The choice of a 40:30:30 effort ratio is thought to lead to adequate estimates in many cases although a more accurate choice could be made using table 4.3. For the Intermediate COCOMO with cost driver multipliers being applied at a component level, the total effort is found by summing the individual component adaptations seperately, ie equation 4.6 now becomes

$$E = E_1 + E_2 + E_3 + \ldots , \qquad E_j = K * a(A_j * L_j^b) \qquad (4.9$$

The feasibility analysis and planning aspects is not covered by the adaptation factor as defined. To overcome this deficiency the adaptation factor should be increased by between 0 and 0.5 according to the amount of conversion analysis and planning needed (see page 558 ref 2 for details). It is also recommended that with the Detailed model the cost drive multipliers should come from the maintenance set rather than the development set.

4.6.2 Developing New Portable Software.

Using the model to estimate costs of adopting portability features requires one to apply the model to the intended software development both with and without the extra requirements for the desired degree of portability. Thus it is necessary to relate the portability characteristics to the parameters of the model. This relation will vary according to the particular development involved, but a general guide to the important parameters in the model can be had by taking each parameter in turn and assessing it against those reuseability characteristics given in section 3.3. A summary of such a general assessment is presented in table 4.7 for the 15 cost drivers, the number of source lines, and the development mode. The main result is the that more design and testing effort will be needed to obtain the the reliability for the portable product. Suppose, for example, that

1. the lines of code goes up 5% to improve robustness then, in the embedded mode, this results in a increase in the effort by a factor of $1.05^{1.20} = 1.06$,

2. the rating for reliability goes from nominal to high, then according to table 4.5 this will involve a 15% increase in effort,

3. some of this extra effort may be offset if say the rating for using software tools can be increased - a rating change from nominal to high gives a 9% gain.

The overall increase is effort is 11%. The purchase of software tools and training to use then fall in the plans & requirements phase, which would have to be estimated separately.

This technique can be refined by applying it at the software component level in the manner described in the previous section.

Table 4.7

SIGNIFICANCE of MODEL PARAMETERS to PORTABILITY ENHANCEMENT

Model parameter	comments
Development mode	unlikely to be affected
Lines of code, L	small to medium increase to improve reliability and robustness, increase due to increased generality
Software reliability	higher reliability usually needed, reflected in more design and test effort
Product complexity	more complex due to higher reliabilty and more generality
Execution time constraint	difficult to cater for in general
Main storage constraint	design may be more complex
Virtual machine volatility	the development environment may change - ex. new tool for testing
Computer turnaround time	may be affected by increased testing
Analyst capability Applications experience Programmer capability Virtual machine experience Prog/language experience	all these personnel related cost drivers may decrease if more capable staff are required for developing the portable software
Use of modern prog/practices Use of software tools	the portabilty need may lead to more use and hence reduce development cost
Required devel't schedule	higher quality (ex. reliability, documentation) will lead to tighter schedule.

CHAPTER 5

IMPLICATIONS IN THE RESEARCH ENVIRONMENT

NOTE this chapter deals less with generalities than with the possible implications of applying Software Engineering techniques to produce portable project documentation (including source text) within the context of the NATS/RSRE contractual arrangement. It is written from a 'researcher's-eye' viewpoint, which accounts for the perceptions of Management's aims, objectives and problems that it contains.

5.1 INTRODUCTION

The degree to which Software Engineering techniques can be applied within a research environment depend upon a number of factors - not the least of these being the extent to which the 'research' can be considered as part of the overall procurement process.

At one extreme, this may consist of a relatively small amount of research effort, which can be considered to be a part, or extension, of the concept/feasibility/project definition phases of the life-cycle model postulated in chapter 2. At this level - where the term 'research' might be considered a little grandiose or inappropriate - it may be as relatively 'easy' to impose some kind of project management discipline on the research team (who may, or may not, be involved in other, later parts of the project) as on other personnel involved in other phases of procurement. To take an (hypothetical) example: a new traffic management facility might entail exploring a new approach to certain operational procedures coupled, say, with a possibility of taking a fresh look at some display techniques. This could represent only a small proportion of the overall requirement and hence it is decided to include it in the accepted management procedures, including the requirement to toe the

line in respect of 'portability' and document control. Predicting the time, and the cost, of even as modest a research task as this is far from easy and may be affected, favourably or adversely, by the degree of real freedom given to the research team regarding the methods they are allowed to use whilst engaged on the job, as distinct from providing some output to be handed on the the next phase. Because it is a fairly restricted and, possibly, well specified piece of 'directed' research, the penalty of applying rigid procurement management methods may not be too high. But, even with this limitation, it may well pay to let the researchers 'do their own thing' until a satisfactory solution is reached and then get them to translate the outcome into a document in accordance with project management criteria.

At the other extreme, it might be envisaged that a major re-structuring of (part of) the ATM system is likely to be required in several years time, involving a serious re-think about a number of inter-related operational policy matters. This could entail exploring a number of different scenarios, any of which could result in a document at the project definition and/or requirement specification stages together with software which could be a significant input to software design. The actual amount of output handed on to successive stages of the procurement exercise may represent only a small proportion of the total research effort entailed and the overheads of applying procurement management techniques throughout all the iterations of the research phase very considerable.

Therefore, with any but the most trivial of 'research' tasks, one is faced with the question:

> 'To what extent does the possibility of producing a reusable (ie portable) module, at any level below the conceptual, justify the (difficult to quantify) extra cost (which may be escalated by repeated iteration) occasioned by enforcing procurement management techniques over the entire research phase, in the interests of making portability a requirement?'

Unfortunately, history is not much help in answering this question. This can probably be attributed to two main factors.

1. Research has been kept well separated from evaluation and procurement, because it has been explicitly understood that it was NOT part of the procurement process.

2. There has been no formal standardization of procurement procedures, below the Contract Management level. Therefore, there has been little commonality between various research and development activities which could lead to exploitation of a mutually accepted 'toolkit' - even if one had been available.

Consequently, portability has not been a requirement at any but the conceptual level.

Nevertheless, examining these factors in the context of past ATC research at RSRE may give some clues as to likely problems in trying to answer the question posed above. Looked at from another point of view, they can be considered as the result of

(1) management objectives, and
(2) available resources (people, hardware, software and methods).

Clearly, these are by no means independent entities.

5.2 MANAGEMENT OBJECTIVES

As was stated above, there has been little commitment, in the past, for research at RSRE to be explicitly considered to be part of the procurement process - at least at the time that the research was initiated. This partly stems from the difficult history of the Linesman/Mediator project, but also reflects the very different nature of the objectives and timescales between what has been under development/implementation and what has been researched. Generally, the former has been relatively imminent and the nature of the desired end-product reasonably well understood. With the latter, however, the aim has been to look beyond what is currently being envisaged, with a degree of speculation very much a function of the likely lead time - somtimes measured in decades. Hence, any 'research' needed as part of the LATCC Development plan has not been done at RSRE and it has only been quite recently that eventual implementation of RSRE output, at any other than the basic conceptual level, has seriously been considered.

(This particularly applies to ATM and System Studies - developments in Radar (eg ADSEL and Monopulse with Cossor, ASR with RACAL-Decca), because of the enormous amount of unique expertise resident at RSRE, seem to have involved a substantially greater de facto input from RSRE than might have been expected from what was said in the previous paragraph. But this work has been very much on the lines of an old-style MoD Contract, arising out of a considerable

amount of on-site research and development, rather than following a logical progression purely under NATS sponsorship. This is not to denigrate taking the opportunity to exploit work being undertaken from a different atarting point, but it is an example which does not lend itself to the overall concept of Project Management suggested in the life cycle structures outlined in chapter 2.)

Hence, it has seemed that any Management objective, from the research phase, would be met with the production of a report (or reports) explaining the basic operational concepts examined, together with experimental design, some implementation details (of the experiment) and a summary of the qualitative and quantitative results. Because of the experimental nature of things, it would be acceptable to examine only a sub-set of the whole, and that the environment used might be restricted and/or partially artificial. Also, to make the best use of limited resources, the experimental design has to be substantially influenced by the available equipment and software. Even accepting all this, it is still possible to conceive that some 'portable' output could be produced, though this might have to be in plain English, an agreed Specification Language (which was the motivation behind the limited study into Decision Tables, and was the intention vis-a-vis aid given to D/DP in responding to any OR arising ot of the DFR demonstration) or, in some cases where only pure calculation is involved, in a common High Level language or Assembly Code. As a result of the continuing machine-dependence of many 'high-level' compilers - especially with regard to input and output - portability of modules involving interaction has, generally, only been possible between identical (or deliberately compatible) systems. To enforce portability between systems that are substantially different would almost certainly entail (possibly severe) restraints on those who generate and those who accept the transported module, because it would be necessary to work to a common sub-set of the facilities available on the two systems.

To accomodate the above state of affairs, at any level below the conceptual, within the procurement process is fraught with difficulty. It would, in principle at least, be far more feasible if an agreed, compatible set of procedures and tools were to be enforced throughout the entire life-cycle of a project. Here, Management is liable to find itself in a dilemma. To integrate the research phase into the project, the procedures to be employed must be fully defined at a very early stage. But, eventually, the implementation will be passed on to a contractor who will, presumably, be chosen by open competition. Unless it

is laid down what tools and procedures will be followed from the outset (which may have the effect of excluding some, otherwise suitable, bids for contract) those used in the early stages, including research, may turn out to be incompatible with those used in the later stages - particularly if the development is spread out over many years. This could have the effect of 'decoupling' the flow of project control at a possibly critical point - making correctness-proving and feedback more difficult.

Additionally, enforcing project management procedures at the research phase will not be without its own costs and these will partly depend upon the scope and probable outcome originally envisaged when that phase is entered. These costs are a measure of the effects of increased restraint on the personnel working on the project and will show up in terms of slower delivery times, possibly a requirement for more staff and, consequently, money. If the end-product is reasonably clear (in which case it scarcely merits the term 'research') then the penalties incurred may be comparable with other (later) stages of the procurement process. But if the outcome, at entry, is 'open-ended' then it would probably be better for the research team to work relatively unrestrained until a identifiable, desirable end-, or by-product emerges. This does not imply that unbridled anarchy should reign supreme - it should still be possible to encourage/coerce, say, the production of standard, structured software. The application of full documentation control, however, is probably best left until a 'production-like' module has been proven and approved.

All this implies, of course, that there is a standard to adhere to! This would have to be accepted by all parties (which, in this case, means all interested NATS Directors as well as RSRE line management), enforced by Management and fit within the context of the available resources. These are considered in the following section.

5.3 RESOURCES

At the end of section 5.1, these were sub-divided into people, hardware, software and methods. These will be considered, briefly, in turn.

5.3.1 People

Over the years, there has tended to be a shift in the backgrounds from which the personnel in the ATC Systems Research Division at RSRE have been drawn. In the early 1970s many people in the teams came from an electronic/system or physics background - often with little or no experience of using computers, even as tools. There was a tendency to pick up sufficient know-how to do the job in hand but, otherwise, to concentrate on the essence of the job rather than on the niceties of the tools themselves. Software Engineering techniques were still at a very exploratory stage and, although there were some attempts to utilize Structured Programming methods and syntax-driven program generators, by and large people did what they could, in their own way, within the limits that the system allowed.

More recently, staff have tended to be recruited from areas where they already have gained a substantial amount of programming experience and some have received formal training in Computer Science or a closely allied subject. It is probably true to say that programming, as a discipline in itself, comes more naturally to these than to many in the first group although, clearly, there will be a substantial overlap between the two groups.

However, whatever the background experience, there has been little requirement to toe any particular line or standard of (eg) documentation, as long as the job has been done. This means that there is a lot of software, which may be more or less idiosyncratic, associated with past and on-going projects for which the question of portability has never arisen. There is also a wide interpretation as to what is meant by 'good' programming techniques. Furthermore, it has to be recognised that to be able to use some of the tools, as they are emerging at present, requires a level of expertise and mathematical ability that is not all that common. Consequently, to introduce a discipline which endeavours to impose some uniform standard must, apart from any in-built resistance to change which results, require a fair amount of education and acclimatisation on the part of those subject to it. The amount of time and effort which would be involved in this alone would not be trivial and would, clearly, have an impact on the system/conceptual progress of a project. The amount of training which any individual might require would depend upon their background, adaptability and

the approach they happen to use at present.

If, subsequently, it transpired that an inappropriate methodology was being employed then the process would have to be repeated. Probably, second time around training would prove easier to the individuals concerned but reaction to the suggestion might be sharper.

How much this situation is likely to change in future is very debatable, considering that there is a projected shortfall of people who are conversant with formal Software Engineering techniques lasting up to the end of the Century. Hence, the need for training - and re-training - is unlikely to go away but should become easier as new staff are recruited into an area where atandards are already established. But it is quite clear that the transition from the present state could not happen overnight.

5.3.2 Hardware And Software

Although it is undisputed that the relative costs of hardware and software are rapidly changing in favour of the former, the cost of hardware is still an item very much to be considered. A 'naked' processor of the type currently in use in the ATC Systems Research Division at RSRE costs £100K - £300K: this represents the irreducible minimum of hardware change that is liable to affect the suitability, or otherwise, of a computer system for a given application. In an ideal world, the processor should be 'transparent' as far as the user's perception of the system is concerned. However, this is still far from being the case - even with such established and highly-used tools as compilers there is still a substantial degree of machine-dependence. This also applies to any current higher-level tools which are aimed at machine assistance for software development.

Things can be more difficult when the question of interfacing with the outside world arises. Even though there is a tendency to home in on a relatively few methods of electrical interconnection (eg RS232, Centronics parallel, IEEE488 etc), there are still considerable gulfs between input and output devices from different manufacturers (but doing similar jobs) when it comes to the communication protocols employed. This situation is changing, however - at least as far

as display is concerned. There are an increasing number of VDU terminals which adhere quite closely to the ANSI standard (eg VT100 and its derivatives) in general alphanumeric character handling. But when graphics are included in the VDU package standards seem to go out of the window! For general graphical displays, packages adhering to some (fairly low) level of GKS are becoming available - though GKS itself has yet to become an ISO standard - and there are an increasing number of products that emulate some version of the Tektronix 4000 series. With input devices, on the other hand, progress towards any sort of standard seems to be much more tenuous, even amongst the two major groups of input devices used in ATM applications today - keyboards and X-Y co-ordinate pointing devices (rolling balls, joysticks, lightpens, pads etc). The possibility of using DVI will probably mean that yet another set of protocol variables will be introduced into an already murky jungle.

How great an effect this diversity may have upon portability depends upon the level of abstraction that can be achieved in describing how any particular input or output device is actually used, or upon how easy it is to reproduce, or even duplicate, devices from one system within another. Frequently, the problems can be overcome by software, though there will often be penalties in change of functional detail, cost, and the extra effort of producing the software. But it is quite likely that considering interaction as part of a portable item will mean that the level at which transportation can be effected will not be lower than that of Pseudo-code or Specification Language.

It is probable that the convergence of standards already occurring with peripheral devices will continue, as there are a large number of independent manufacturers who wish to be compatible with the main computer manufacturers' hardware. (But it is debatable whether the current trend towards greater quantities of 'local intelligence' will prove to be a help or a hindrance towards achieving commonality). With the computer companies, however, commercial pressures may act in the opposite direction, as part of the strategy to hang on to a 'captive' customer base. Hence, it is likely that some machine dependence will persist, until a change is enforced by customer pressure.

What has been said about hardware is still, largely, true of software. There has been little incentive for different computer manufacturers to work

to common standards, though there has been some pressure from major end customers (eg MoD) to establish standards (eg CORAL66, MASCOT) when setting up procurement contracts with major contractors. This is becoming reflected in what some computer manufacturers currently offer (often by using packages developed by separate software houses and, thereby, possibly proving difficult to establish responsibility for maintenance). The momentum of this trend should be enhanced with the advent of Ada, and there is an increasing de facto pressure from the independent peripherals manufacturers regarding such developments as GKS and the OSI networking standards. But progress is still lamentably slow and it would appear likely to remain so for some considerable time, since a large part of the computer industry's market - either directly or via OEMs - is still in 'turnkey' systems where the supplier to the end user has only a limited interest in compatibility with competitors' products.

Against this, it must be acknowledged that some independent producers of software development aids are anxious to promote the sale of their products to users of different hosting computers. It is quite conceivable that some of these may prove to be viable, stable products, with a wide application, but none are outstanding in this way at the moment. Again, development of a standard Programming Support Environment, such as the APSE, could accelerate progress in the development of a stable and compatible set of program development tools.

As far as the current state within the ATC Studies Division at RSRE is concerned, the machine dependence of compilers (already referred to) also applies to the few other, rather limited, software production aids available. Consequently, any software developments in the foreseeable future are bound to have a strong VAX-oriented dialect, modified by the particular array of input/output devices and protocols utilized on any particular project. To change this state of affairs will call for very careful examination of tools, many of which are still emerging from a rudimentary state, and a substantial investment in money and man-months (?man-years). A possible alternative (or pre-cursor) to this is to consider whether there are any methods which can be adopted, within the current constraints, which could result in an improvement in portability.

5.3.3 Methods

As has been pointed out, one of the main problems of applying formal, procurement-type techniques in research is that the end product can be very unclear when a particular project begins. This tends to mean that it is impossible to start with anything but a very loose 'specification'. This, in turn, gives rise to the conclusion that to integrate the activity into a continuous, production-like process, using rigidly applied formal methods, may prove very difficult and counter-productive. Nevertheless, it is clearly desirable that some control should be possible for a reasonable transfer of information, both from the research team to those who follow after and also between members of that team working on different parts of the project. The question is, what is meant by 'reasonable'.

It can be assumed that, early on in the project, a general outline of what is required will be available. Thus, it may be possible to draw some kind of network or activity diagram representing the likely components required in the study. Some of these may be well defined, others much less so. It may be possible to treat some as 'black boxes' which are independent in their own right; others may appear to involve substantial interaction with other components. This, in itself, introduces some diversity into the interpretation of 'reasonableness' of documentation standards applied in the early stages of a project. At the program writing (code production) level it should be possible to establish pretty firm guidelines regarding structuring and internal documentation. However, there might be more difficulty in maintaining true synchronism between this and the 'software design' level above, or the 'system specification' level above that. There would probably be times when these various levels could all be 'up to date' (ie in step with one another), but there would almost inevitably be periods when there would be some disparity between levels - at least in the case of some of the less well-defined components. This is because there could be 'refinement' going on at the design and programming levels which might tend to be moving towards the understood specification or, alternatively, demonstrating where that specification was in error. To apply full documentation control at all levels under these conditions could be very time consuming, frustrating and inefficient (and could, incidentally, absorb the effort of one, or more, dedicated persons).

However, if a more flexible approach is used it would have to be on the understanding that time would be allowed, AND USED, for documentation to be brought fully up to date when recognizable milestones are reached.

With this in mind, it is worth noting that the team working the largest single project sponsored by NATS at RSRE has already adopted a more formalised approach to documentation standards and that others are being encouraged to follow suit.

So far this section has concentrated on documentation control, because this is the area for which most tools are available and which is best understood. It probably also calls for the lowest initial financial outlay. Another control aspect for portability is that of Verification and Validation. No tools are available for either of these within the RSRE team (apart from those associated with compilation) and it looks as though 'manual' methods will remain the only option for some time to come. However, it is not impossible that limited post hoc correctness-testing processes (eg SDS) could be made available but it is probably only worth using these when the detailed requirement is thoroughly understood on a product likely to go into production. This has not been the case so far - the nearest attempt, to date, being represented by the DFR experiment.

5.4 WHAT HAPPENS ELSEWHERE?

It is perhaps worth mentioning at this point that what has been said about previous practice in the ATC System Research Division at RSRE is quite typical of the Establishment in general.

Although projects which have been monitored by RSRE applications divisions may have arisen out of basic work initiated on site, the amount of software transferred directly into the field has been very small. This is in spite of MoD vetting Contractors to see that they conform to Quality Management (DEF-STANs 05/29 & 05-21) and Documentation (JSP188 - ref 21) standards. The reasons for this are very similar to those outlined above - namely, that research work has been done with a view to making an economical assessment of an idea with no requirement to produce robust, operational-quality products. In fact, experience has often been that Contractors have shown

extreme reluctance to take on board anybody else's software and there has been no insistence by prime Customers (usually one of the armed Services) that they should. Hence, once again, portability has been at the Conceptual or Specification level, depending on the degree of RSRE involvement in the development and procurement process.

There are odd exceptions to this in which the story has been almost diametrically opposed to what has just been said. In these cases, an outside firm has taken on a finished product lock, stock and barrel. ALGOL68 and SDS were two such products. But with other RSRE generated items in the software area, such as CORAL66 and MASCOT, the product was handed over at the Definition level.

5.5 CONCLUSIONS

From earlier sections it can be seen that the concept of of portability (re-usability) can be envisaged at a number of levels. At its most abstract, it can be effected in the form of an idea expressed in plain (and, hopefully, unambiguous) English. At the other extreme, it can be in the form of source text or object code which has been (or can be) subjected to rigorously controlled design, quality management and fully 'layered' standard documentation. The difference between these extremes can represent a many-times increase in cost and development time. Also, as increasing rigour is applied, the level of staff awareness of software engineering tools and methods (possibly calling for higher levels of specialized mathematical ability) will have to rise. This means that there will be a requirement for further training. There may also be a need for extra staff - if only for documentation purposes.

To impose this on any real research activity would almost certainly be counter-productive for most of the time in that progress would be inhibited, staff morale depressed and superfluous documentation produced in ever-increasing quantities. Therefore, it appears better to allow research personnel to work with relatively mild constraints, whilst they are busy 'behind the scenes' and only apply much firmer control when an identifiable, portable module emerges. The motivation to adhere to much stricter control should then be much higher, because there would be the prospect of a piece of research work being taken further along the production path. This is not so say that nothing should ever be altered from current practice. There are areas where some greater adherence to standards could bring benefits, without imposing any great strain on current procedures.

IMPLICATIONS IN THE RESEARCH ENVIRONMENT

Any modification to the way research is currently carried out which would increase the overheads on more than a small proportion of the various activities involved would have to be sanctioned by all the appropriate sponsoring Directorates, preferably in a way which shows where the extra cost should fall.

CHAPTER 6

SUMMARY AND CONCLUSIONS.

6.1 SUMMARY

A considerable amount of effort is being put into software engineering by governments, industry and academiae to alleviate the so called 'software crisis' of the demand for software outstripping the supply of trained personnel needed to produce that software. That software reuse is potentially one of the most important weapons in warding of this crisis was confirmed in a quantitative manner by application of software cost modelling techniques.

To understand why the full potential of reusing software has not been achieved requires an appreciation of the software development process, the characteristics of reusable software, the problems of trying to reuse software and the economics involved.

The software development process was described in terms of the classical life cycle models, and software was defined as the products of the life cycle phases. This is a very important definition since the term software now encompasses such items as design specification documents and is not just limited to program code. In fact a case study was used to show why in many instances the code is the least likely software to be reused even when the nominal functionality might suggest otherwise. On the other hand it has to be acknowledged that reusable software at the code level is a common feature of certain tools such as compilers and operating systems, albeit reusablity then frequently relies on an underlying hardware commonality.

Of the the various characteristics of reusability that were discussed, modularity and adoption of widely accepted standards were seen as vital to the production of reusable software. Modularity at both functional and design levels is needed to provide software in the form of components

suitable for later retrieval. The interface of a component must be specified in a formal language so that components can be fitted together subsequently. This plug and socket point of view of 'how' components fit together is sometimes referred to as the syntactic abstraction to distinguish it from the semantic abstraction of 'what' the components should do. Other important characteristics are understandabilty, reliability and maintainability and can be achieved using good software engineering practices, tools, and methods. Usually higher quality will be needed for software developed with reuse in mind and the associated higher cost will be offset by subsequent reuse. How this higher cost can be estimated was discussed with the use of a cost model. The model was also used show how the cost of adaptation of existing software and the cost of maintenance can be estimated.

Software developed in a research environment was shown to fall roughly into two categories. In the first, the software is almost incidental to the main aim of producing and exploring concepts and consequently, in general, little effort is made or needed to make the software reusable. Much of the work in the Air Traffic Control Studies division can be considered to fall in this category. In the second category, software is seen as a major deliverable of the research programme and effort is made to ease subsequent reuse. Research into software tools is a good example in this category. Even so, it is normal practice to put the research software out to contractors to ruggedise and maintain the product. The cost models and life-cycle models of the type discussed in the text are not designed for application to software produced in a research environment, although they can applied there to some extent but the results obtained must be treated with caution.

However, it still has to be recognized that there remains a distinct reluctance on the part of producers of software as 'engineered' products to accept even reasonably fully developed modules from outside their own organisations - even when it entails re-working a substantial amount of code design and implementation. This is an attitude that will take a substantial amount of customer pressure and building-up of confidence in the quality of other peoples' production techniques to dispel.

6.2 CONCLUSION AND RECOMMENDATIONS

Lack of resources - financial, manpower, and time - and the observed repeated writing of similar software are driving the push for more reuseable software. The key features of reusable software are that the components should have widely applicable functionality and should be specified and produced with the aid of widely acceptable standards. The software engineering community is not always able to be very specific in advising on how best to develop reusable software, not least because methods and tools are still evolving and subject to research. However the following set of idealised guide lines can be recommended to developers and procurers:

1. Adopt organisation-wide standards in methodologies, tools and methods.

2. Choose methodlogies that use formality in as many phases as possible.

3. Choose tools and methods that integrate together. Avoid having different tools and methods doing the same job. Adopt structured programming techniques. Use one high level language and only permit assembly and machine code in exceptional circumstances.

4. Make more use of s/w cost estimating models. Keep in-house statistics to provide input data applicable to your own situation.

5. When preparing Requirement Specifications, call for statistical data gathering facilities, preferably automatic, to support project control, monitoring of reliability and maintenance, and further input for cost modelling analysis.

6. Don't expect miracles! Be prepared for mistakes to occur and some adverse staff reaction. Adopting organisation-wide standards will take time, education, training and money.

7. Co-ordinate with other bodies to increase the commonality of standards and their adoption.

8. Keep abreast of the rapidly moving software engineering field. Read assessments of leading tools and methods. Be aware of the trends in Integrated Programming support environments, Ada, formal and semi-formal methods, and IKBS techniques.

9. Study published assessments of leading tools and methods, being prepared to try something new if it appears to suit your purposes better or plugs an unfilled gap. Keep statistics in order to make a proper assessment of the cost effectiveness of any new tool or method.

10. Get the backing of all aspects of management to adopt, promulgate, and enforce standards.

Obviously these guide lines have to be interpreted in light of budgets, timescales and existing commitments, but it should be remembered that, for systems with a large amount of software, the cost of maintenance dominates the overall life-cycle costs and capital investment in suitable tools and methods will significantly reduce the software bill.

In particular, NATS might consider adopting military standards JSP188, Ada etc.

Reusability is not normally a strong requirement of the software produced in the research environment such as that of the Air Traffic Control Systems Research division in RSRE and the following procedures are suggested:

1. Discuss with the sponsor(s) of the research the degree of reusability expected and the area where reuse is forseen.

2. Determine the standards to be applied, particularly those of documentation. Give due account to the standards already in use in both the research and the potential reuse area, and to the resources required.

3. Monitor and review standards at agreed milestones, and as the situation demands.

4. Use firm research management to ensure standards are kept.

5. Be prepared to adopt more tools and methods.

Where the reusable software is the main output of research these procedures need reinforcing, making using the previous sets of guide lines.

The Air Traffic Control Systems Research division has carried out a brief assessment of its past and present procedures relating to reusability of software, especially in regard to documentation, and is currently producing a set

of internal documentation standard guidelines. It has also started using a software tool for configuration management of software modules. In view of MOD policy requiring Ada for new software after 1987, there is likely to be an increasing pressure to consider moving from CORAL to Ada. This pressure will be assisted by the trends to more formal methods for improving verification and to the increased use of integrated support environments.

CHAPTER 7

REFERENCES.

1 B.W.Boehm. Software and its impact: A quantitative assessment. Datamation Vol 16, No 3, April 1973.

2 B.W.Boehm. Software Engineering Economics. Prentice-Hall, Englewood Cliffs, N.J, 1981.

3 NEDO. Software Engineering and CADMAT. Electronic Capital Equipment EDC, National Economic Development Office, 1984.

4 J.Morrisey and S.Y.Wu. Software Engineering: An Economic Perspective. Proc 4th International Conference on Software Engineering, Sept 1979, pp 412-422.

5 DTI. The STARTS (Software Tools for Application to Large Real Time Systems) Guide. Department of Trade and Industry, 1984.

6 J.C.Batz, P.M.Cohen, S.T.Redwine and J.R.Rice. The application-specific task area. IEEE Computer, Vol16, no.11, Nov 1983, pp 78-85.

7 J.G.P.Barnes. Programming in Ada (2nd edition). Addison-Welsey, 1984.

8 J.Nissen and P.Wallis (Editors). Portability and style in Ada. Cambridge University Press, 1984.

9 K.Iwamoto and O.Shigo. A software design system based on a unified design method. Computer science and technologies - Japan annual review in electronics, computers and tele-communications. OHM - North-Holland 1982, pp 110-123.

REFERENCES.

10 G.Roman, M.J.Stucki, W.E.Ball and W.D.Gillett. A total system design framework. IEEE Computer, Vol.17, no.5, May 1984, pp 15-26.

11 B.A.Kitchenham and N.R.Taylor. Software cost models. ICL Technical Journal, Vol 4, Issue 1, May 1984, pp 73-102.

12 R.Yeh. Software engineering. IEEE Spectrum, Vol.20, no.11, Nov 1983, pp 91-94.

13 B.W.Boehm. Software engineering. IEEE Trans Computing, Vol C-25, no 12, Dec 1976, pp 1226-1241

14 DoD MIL-STD-1521A. Technical reviews and audits for systems, equipments, and computer programs, Department of Defence (USAF), Washington DC, June 1976.

15 P.W.Metzger. Managing a Programming Project, Prentice-Hall, Englewood Cliffs, N.J., 1973.

16 P.Freeman. Tutorial on software design techniques, IEEE CS Press, Los Alamitos, Calif., 1976.

17 M.Stanley. Software cost estimating, RSRE Memo no 3472, 1982.

18 L.H.Putman and A.Fitzsimmons. Estimating software costs. Datamation 1979, Sept pp 189-198, Oct pp 171-178 and Nov pp 137-140.

19 Distributed system design technology, NATO defence research group, AC/243 (Panel III/RSG 14), 1984.

20 Special issue on software reusability, IEEE Trans. Software Engineering, Vol SE-10, no.5, Sept. 1984.

21 MoD. Specification for the Documentation of Software in Real-Time Computer Sytems. Joint Service Publication (JSP) 188, 2nd Edition, April 1977

22 T.C.Jones. Reusability in Programming: A survey of the State of the Art, opus cit.(ref 20), pp 488-494.

23 P.Wegner. Capital-Intensive Software Technology. IEEE Software, Vol. 1, no. 3, July 1984, pp 7-45.

24 M.Shaw. Abstraction techniques in modern programming languages. IEEE Software, Vol. 1, no. 4, Oct. 1984, pp 11-26.

CHAPTER 8

GLOSSARY

8.1 GLOSSARY OF LESS WELL KNOWN TERMS

ATC Air Traffic Control.

ATM Air Traffic Management. The management of a large and varied mix of air traffic by a co-ordinating ground based ATC authority. ATM and System studies examine possible ways of exploiting limited resources, such as airspace, aircraft performance, airports and ATC procedures, in the context of changing traffic demands up to and beyond the year 2000.

ADSEL The forerunner of the Secondary Radar Mode S.

ASR Airfield Surveillance Radar.

Coral66 The MoD standard high level programming language.

DFR Departure Flow Regulator. RSRE was asked to develop an experimental demonstration aid aimed at helping Air Traffic Controllers integrate aircraft taking off into the traffic flying out of the United Kingdom airspace.

D/DP Directorate of Data Processing.

DVI Direct voice input.

GKS Graphics Kernel System. A proposed international software standard for defining the control of graphics.

LATCC London Air Traffic Control Centre.

MASCOT Modular Approach to System Construction and Test. A notation for describing software components, and a set of tools for assembling, testing, and synchronising them.

GLOSSARY

NATS National Air Traffic Services.

OR Operational Requirement.

SDS Software Development System. A database tool intended to capture the large amount of information associated with project software development.

VDU Visual display unit.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 3884	3. Agency Reference	4. Report Security Classification UNCLASSIFIED
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS AND RADAR ESTABLISHMENT		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location CIVIL AVIATION AUTHORITY		

7. Title

REUSEABLE SOFTWARE

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
SIMCOX L N	BALL R G			

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract The issues involved in reusing software are examined both in a general context and in the context of a research environment. In the general context, the characteristics of reusable software are discussed and the views of both the potential developer and those seeking to reuse existing software are considered. An assessment of reusable software is carried out using the concept of the life cycle model, the economic aspects being based on the COCOMO parametric software cast model. General recommendations for improving the portability situation are made for those involved in procuring, developing, and researching software.

END

DTIC

7-86